2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems

Using CSP to Model and Analyze TinyOS Applications

Allan I. McInnes
Electrical and Computer Engineering
University of Canterbury
Christchurch, New Zealand
e-mail: allan.mcinnes@canterbury.ac.nz

Abstract—**The TinyOS concurrency model, although easier to reason about than shared-state threads, may still produce undesirable behavior as a result of unexpected interleaving of concurrent activities. This is problematic, since TinyOS applications are typically intended to run unattended for long periods of time, and must be reliable. In this paper, we describe a technique for modeling the interactions between TinyOS application components, and between an application and the TinyOS scheduling and preemption mechanisms, using the process algebra CSP. Analysis of the resulting process models can help TinyOS application developers to discover and diagnose concurrency-related errors in their designs that might otherwise go undetected until deployment of the application.**

Keywords*—*TinyOS; Concurrency; Process algebra.

I. Introduction

TinyOS [1] is a popular operating system for embedded sensor networks. TinyOS and the applications that run on it are written in nesC [2], a dialect of C that adds support for a component-based programming model. TinyOS applications are constructed as a graph of components that interact both with each other, and with the underlying TinyOS scheduler and platform hardware. Ensuring that all of these interactions occur correctly is an important aspect of developing a reliable TinyOS application.

Gaining an understanding of how TinyOS application components interact with each other is complicated by the inclusion of concurrency within the TinyOS model of computation [3]. Concurrency is introduced into TinyOS by interrupts, which can cause one component to preempt the execution of another, and by tasks, which are a way for components to defer the execution of operations that are not time-critical. Care has been taken in the design of TinyOS and nesC to reduce the likelihood of concurrency-related errors, for example by including race-condition checks in the nesC compiler, and using a run-to-completion execution model for tasks. Nevertheless, interrupts and tasks both introduce the possibility of unanticipated interleavings of different activities, which may cause undesirable application behavior. For example, consider the following snippet of nesC, which is intended to repeatedly execute an activity by using a task to start the activity again each time completion of the previous execution of the activity is signaled:

```
command void RepeatedActivity.stop()
{ call Activity.stop(); }

event void Activity.done()
{ post next(); }

task void next()
{ call Activity.start(); }
```

The cycle of repeated executions will usually stop when `RepeatedActivity.stop()` is called, because stopping the activity will prevent the `Activity.done()` event that triggers a new execution from being signaled. However, should `RepeatedActivity.stop()` be called after the `next()` task has been posted to the scheduler, but before it has been executed, the execution cycle will not stop: later execution of `next()` will restart the cycle even though the `RepeatedActivity.stop()` command has been executed. This erroneous behavior results from an unanticipated interleaving of the concurrent stop and restart activities.

In this paper, we show how TinyOS application developers can shed light on potential concurrency-related problems in their application designs by using the process algebra CSP (Communicating Sequential Processes) [4]:

- We define a mapping from nesC programs to CSP process models (section III-B). This mapping formalizes the component interactions within a nesC program. As part of the mapping, we define several processes that ease the translation from nesC to CSP, by providing a nesC-like syntactic sugar over the underlying process model (section III-C).
- We develop a CSP model of TinyOS task-scheduling and preemption (section III-D). This model clarifies the execution semantics of TinyOS, and allows interactions between TinyOS applications and the TinyOS concurrency mechanisms to be analyzed.
- We demonstrate how our nesC to CSP mapping and model of the TinyOS concurrency mechanisms can be used together to model and analyze TinyOS applications using off-the-shelf tools (section IV).

The CSP approach described in this paper offers several features not found in previous approaches to analyzing TinyOS applications, which we discuss in section V.

978-0-7695-3602-6/09 $25.00 © 2009 IEEE
DOI 10.1109/ECBS.2009.34

II. TinyOS and nesC

TinyOS is an embedded operating system developed to meet the needs of programming resource-constrained embedded networks [1]. It consists of a large collection of components that implement services useful in networked embedded systems, such as communications, timing, and lightweight task scheduling. TinyOS is not a hard real-time operating system, but provides a component-based, event-driven programming model within a very small footprint.

The nesC language was created to support TinyOS [2]. Reflecting TinyOS' component-based philosophy, nesC programs are structured as compositions of interacting components. These components come in two flavors: *modules*, which implement specific functionality, and *configurations*, which define connections between components.

Each nesC component has one or more *interfaces*, which consist of *commands* and *events*. Commands are used to request that a component perform some service, while events are a mechanism for signaling to a component that an activity it requested has been completed, or that an external phenomenon such as an interrupt has occurred. A component that *provides* an interface must implement functions for each command in the interface, and may signal any of the events, while a component that *uses* an interfaces must implement functions to handle each event in the interface, and may call any of the commands. Configuration components connect ("wire") interface providers to interface users.

Again reflecting the needs of TinyOS, the behavior of nesC programs is largely event-driven. Hardware interrupts can trigger events, the effects of which propagate up the component graph as a cascade of further events. Event-handling functions may also make command calls back down the component graph as part of the overall system response to the initiating event. In situations where the response to an event is not time-critical, the overall responsiveness of the application may be improved by deferring execution of the response until the processor is idle. Deferred execution is achieved by posting a *task* that implements the activities to be deferred.

TinyOS includes a simple task scheduler that maintains a FIFO queue of pending tasks, which are executed whenever the processor is not occupied responding to an interrupt. The scheduler is non-preemptive, and executes each task in a run-to-completion manner. Operations executed by the scheduler are referred to as *synchronous*. This style of execution makes tasks atomic with respect to each other, and thereby prevents data races between tasks. However, task execution can be preempted by interrupt-triggered *asynchronous* events, so nesC also includes the capability to define *atomic* code blocks which are protected from preemption.

The concurrency introduced by tasks and interrupts increases the difficulty of making TinyOS applications reliable, because developers must anticipate all of the possible interleavings of concurrent activities. The reliability of TinyOS applications can be improved prior to deployment by subjecting the application to simulation via TOSSIM [3], unit testing [5], and perhaps lab testing. But uncovering concurrency-related errors via testing and simulation is a difficult task, due to the timing sensitivity of many concurrency errors. Furthermore, even if an error is found during testing, pinning down the actual source of the problem can be a time-consuming process. It's therefore worthwhile to consider other options for analyzing the concurrent behavior of TinyOS applications.

III. CSP Models of TinyOS Applications

Model-checking is a technique for discovering errors in a design by systematically exploring the state-space of an abstract model of the design [6]. In order to apply model-checking techniques to TinyOS applications, it's necessary to abstract the application design into a formal model that is amenable to model-checking. We have opted to use the process algebra CSP as our modeling formalism, since its characteristics are well-matched to those of nesC and TinyOS:

- TinyOS applications are concurrent; CSP is designed to model concurrent systems.
- TinyOS applications are event-driven; CSP is an event-based formalism.
- TinyOS applications are hierarchical compositions of components; CSP process models are hierarchical compositions of processes.
- TinyOS applications are composed by wiring together interfaces; CSP processes are composed by wiring together channels.

We build on the intuitive correspondence between nesC programs and CSP processes to develop a systematic mapping from nesC to CSP, and add models of the TinyOS task-scheduler and preemption rules to construct CSP processes that model TinyOS applications.

A. Introducing CSP

CSP [4] is a mathematical theory of concurrency and interaction, in which concurrent systems are modeled as collections of event-transition systems (processes) that interact by synchronizing on shared events. CSP events are abstract symbolic representations of interactions. For example, a model of an online purchase might include events that represent ordering an item, confirming the order, providing payment, and shipping the ordered item. Interfaces between processes can be defined using channels which carry values of some specified type; each occurrence of a value being passed through a channel corresponds to a single event.

Simple sequential processes are defined by using the prefix operator $\rightarrow$ to specify sequences of events, e.g.

$$\begin{aligned}&\mathit{OnlinePurchase} = \\ &\quad \mathit{order}?\mathit{item} \rightarrow \mathit{confirm}!\mathit{item} \rightarrow \\ &\quad \mathit{request\_payment}!\mathit{cost}(\mathit{item}) \rightarrow \\ &\quad \mathit{receive\_payment}?p \rightarrow \mathit{ship}!\mathit{item} \rightarrow \mathit{SKIP}\end{aligned}$$

where ? and ! indicate channel input and output respectively, and *SKIP* is a primitive process representing successful termination. Observant readers will have noted that the online purchase process has the unfortunate habit of shipping an item regardless of the amount of payment received. Fortunately, the CSP process notation also provides a variety of other operators for defining behaviors, such as:

- Conditionals (if $p = \mathit{cost}(\mathit{item})$ then ... else ...)
- Alternatives ($\mathit{DisplayCart} \,\Box\, \mathit{DisplayCatalog}$)
- Nondeterministic outcomes ($\mathit{Transaction} \sqcap \mathit{Error}$)
- Process sequence ($\mathit{Server}(x) = \mathit{Login};\ \mathit{Hello};\ \ldots$)
- Parallelism ($\mathit{Servers} = \mathit{Server}(1) \,|||\, \mathit{Server}(2)$)
- Interfaces ($\mathit{Customer} \,[\![\, \mathit{OrderEvents} \,]\!]\, \mathit{Servers}$)

CSP includes a rich theory of process refinement and equivalence based on analyzing the sequences of events that processes can be observed to perform. The FDR2 model-checker [7] is an industrial-strength analysis tool that can be used to automatically check CSP models for properties such as deadlock or livelock, and to evaluate process models for conformance to specifications.

B. Mapping nesC to CSP

Since we're interested in finding errors in component interactions within a TinyOS application, our mapping from nesC to CSP focuses on modeling the execution of nesC commands and events. We model nesC components as CSP processes that interact through CSP events which represent nesC commands, events, and task control functions.

1) CSP Event Structure: The CSP events that represent nesC commands and events are compound symbols which encode information about the command or event, the interface and component with which it is associated, and the execution context (synchronous or asynchronous). For example, initiation of the `read()` command from the `Read` interface by a component `SenseC` during synchronous execution is represented by the event *exec.begin.SenseC.Read_read.sync*.

Although it's tempting to map execution of a nesC command or event directly to a single CSP event, such a mapping would not allow us to accurately represent the behavior of command/event execution. Commands and events in nesC are both compiled down to C functions [8]. Thus a command that calls another command will not complete its execution until the second command finishes executing, and control returns to the first command. To reproduce the behavior of function calls in our CSP model, we represent command/event execution as a pair of events that indicate the beginning and end of an execution.

In general then, nesC commands and events are represented by the CSP events associated with the *exec* channel:

$$\textsf{channel}\ \mathit{exec} : \mathit{Exec}.\mathit{NesC\_Function}.\mathit{SyncType}$$

where the types

$$\begin{aligned}&\textsf{datatype}\ \mathit{Exec} = \mathit{begin} \mid \mathit{complete} \\ &\textsf{datatype}\ \mathit{SyncType} = \mathit{sync} \mid \mathit{async} \mid \mathit{hw\_async}\end{aligned}$$

contain symbols that encode the execution step and context. The use of *SyncType* will become clear when we introduce modeling of interrupt preemption in section III-D.

The *NesC_Function* type that appears in the definition of *exec* is a set that depends on the TinyOS application being modeled. It contains compound symbols that encode the component and interface of each function (command or event). We encode interfaces as types that combine the interface name with the name of each function in the interface using the format *<interface-name>_<function-name>*, as in the example in Fig. 1. The full interface of a component is a type that is a union of the individual interfaces the component provides and uses, prepended with the component name to model the component-local namespaces used in nesC [8]. In the case of interfaces that are renamed within the component, we incorporate the new name into the prepended component name (see Fig. 1). Finally, the *NesC_Function* type is the union of all component interfaces. For example, for an application containing the components `MainC`, `SenseC`, `TimerC`, and `SensorC`, the corresponding *NesC_Function* is:

$$\begin{aligned}&\textsf{nametype}\ \mathit{NesC\_Function} = \\ &\quad \bigcup\{\mathit{MainC\_IF}, \mathit{SenseC\_IF}, \mathit{TimerC\_IF}, \mathit{SensorC\_IF}\}\end{aligned}$$

We model task-related operations using a style similar to that used to model commands and events. Posting of tasks to the scheduler is modeled by events associated with the *task_post* channel, while the initiation and completion of task execution are modeled as *task_exec* events:

$$\begin{aligned}&\textsf{channel}\ \mathit{task\_post} : \mathit{Task} \\ &\textsf{channel}\ \mathit{task\_exec} : \mathit{Exec}.\mathit{Task}\end{aligned}$$

where *Task* is a datatype specific to the application being modeled. The *Task* datatype consists of symbols that represent each task in the application, defined using the form *<component-name>_task_<task-name>*. For example, the task symbol for a module `SensorC` with task `senseResult()` would be *SensorC_task_senseResult*.

2) Modules: We model modules as processes that provide a selection of behaviors corresponding to the functions implemented by the module. By default, functions in TinyOS are assumed to be synchronous, which means that they cannot be called during handling of an asynchronous event like

```
interface Read<val_t> {
  command error_t read();
  event void readDone(...);
}

module SenseC {
  provides interface SplitControl;
  uses interface Read<uint16_t>;
  uses interface Timer<TMilli> as MTimer;
} ...
```

$$\begin{aligned}&\text{datatype } \mathit{ReadIF} = \mathit{Read\_read} \mid \mathit{Read\_readDone}\\&\text{datatype } \mathit{SenseC\_IF} =\\&\quad \mathit{SenseC}.\bigcup\{\mathit{ReadIF}, \mathit{SplitControIF}\}\\&\quad \mid \mathit{SenseC\_MTimer}.\mathit{TimerIF}\end{aligned}$$

Figure 1. Datatypes for interfaces

an interrupt. In contrast, functions labeled with the `async` keyword can be called from an asynchronous context [9]. To maintain this distinction in our CSP model, we split the module process into synchronous and asynchronous parts. The synchronous part is defined as an external choice ($\Box$) over commands, events, and tasks, since we assume that only one of these can be executing at a time. The asynchronous part is defined as a parallel composition of processes that represent individual command or event-handling functions. The module model itself is then a parallel composition of the synchronous and asynchronous parts.

The general form of module process models is:

$$\begin{aligned}&\mathit{Module\_{<}component\text{-}name{>}} =\\&\quad \text{let}\\&\qquad \mathit{SyncFunctions} =\\&\qquad\quad \mathit{<command\text{-}1\text{-}behavior>};\ \mathit{SyncFunctions}\\&\qquad\quad \Box \cdots\\&\qquad\quad \Box\ \mathit{<event\text{-}N\text{-}behavior>};\ \mathit{SyncFunctions}\\&\qquad\quad \Box\ \mathit{<task\text{-}N\text{-}behavior>};\ \mathit{SyncFunctions}\\&\\&\qquad \mathit{AsyncFunctions} =\\&\qquad\quad \text{let}\\&\qquad\qquad \mathit{AsyncF1} =\\&\qquad\qquad\quad \mathit{<async\text{-}cmd\text{-}1\text{-}behavior>};\ \mathit{AsyncF1}\\&\qquad\qquad \cdots\\&\qquad\qquad \mathit{AsyncFN} =\\&\qquad\qquad\quad \mathit{<async\text{-}event\text{-}N\text{-}behavior>};\ \mathit{AsyncFN}\\&\qquad\quad \text{within}\\&\qquad\qquad \mathit{AsyncF1} \mathbin{|||} \cdots \mathbin{|||} \mathit{AsyncFN}\\&\quad \text{within}\\&\qquad (\mathit{SyncFunctions} \mathbin{|||} \mathit{AsyncFunctions})\end{aligned}$$

As the examples in section IV show, for simple module behaviors, the state of a component can be maintained by arranging the synchronous part of the module process model as a state transition system. The same technique can be used to constrain the states in which certain functions are available, which makes it possible to specify the acceptable sequences of function calls as part of the module behavior. The resulting state transition system resembles an interface automaton [10], and, like an interface automaton, can be used to check component interface compatibility. In situations where a state transition system is insufficient to model the module state, local processes representing variables can be added to the module process model.

Within a module process, we model the behavior of an individual function f as the process

$$\begin{aligned}\mathit{FnDef}(f, \mathit{Body}) = \mathit{exec.begin.f.sync} \rightarrow \mathit{Body}\ ;\\ \mathit{exec.complete.f.sync} \rightarrow \mathit{SKIP}\end{aligned}$$

where *Body* is a process defining the actions performed by the command or event. Note that *FnDef* provides a model for synchronous functions only, since its events are postfixed with the symbol *sync*. Since asynchronous functions can be called from both synchronous and asynchronous contexts, we require a slightly more complex model to properly capture their behavior. The process

$$\begin{aligned}&\mathit{AsyncFnDef}(f, \mathit{Body}) =\\&\quad \mathit{exec.begin.f?s} \rightarrow\\&\qquad (\mathit{Body}\ ;\ \mathit{exec.complete.f.s} \rightarrow \mathit{SKIP})\end{aligned}$$

is similar to *FnDef*, but keeps track of the *SyncType* of the initiating event.

In general, the actions performed in the *Body* of a function will be one or more calls to other functions. We use CSP sequential composition ($;$), conditionals (if ... then), and recursion to capture control-flow, and model function calls with the process

$$\begin{aligned}\mathit{FnExec}(f, s) = \mathit{exec.begin.f.s} \rightarrow\\ \mathit{exec.complete.f.s} \rightarrow \mathit{SKIP}\end{aligned}$$

where s is the *SyncType* of the function call, and indicates the context of the call. Synchronizing *FnExec* with a corresponding *FnDef* process has the effect of triggering execution of the function f, and blocking the *FnExec* process until the body of the *FnDef* has completed. Blocking *FnExec* in this way provides the desired behavior for nested command and event executions.

The behavior of tasks is modeled in a fashion similar to that used to model commands and events, as a task body surrounded by beginning and completion events:

$$\begin{aligned}\mathit{task}(t, \mathit{Body}) = \mathit{task\_exec.begin.t} \rightarrow \mathit{Body}\ ;\\ \mathit{task\_exec.complete.t} \rightarrow \mathit{SKIP}\end{aligned}$$

Unlike commands or events, task execution is triggered by the scheduler (section III-D). Tasks are added to the scheduler by posting them, an action modeled by the process:

$$\mathit{post}(t) = \mathit{task\_post.t} \rightarrow \mathit{SKIP}$$

3) Configurations: We model the inter-component connections defined by a configuration as an interface-parallel composition of component processes synchronizing on events in their connected interfaces. However, because *exec* events are structured to provide each module with its own namespace (section III-B1), modules do not by themselves have any events in common. We therefore use the CSP renaming operator to map events associated with one module to events associated with the other module. Renaming can also be used to map the interfaces inside a configuration to those provided by the configuration.

The wiring of components *A* and *B* to each other is modeled by the process

$$\begin{array}{l} wiring(A, B, Connections) = \\ \quad \text{let} \\ \quad\quad SharedIF = \{exec.e.IFa.f.s \\ \quad\quad\quad\quad\quad\quad \mid (IFa, \_, Fns) \in Connections, \\ \quad\quad\quad\quad\quad\quad\quad e \in Exec, f \in Fns, s \in SyncType\} \\ \quad \text{within} \\ \quad\quad (A \\ \quad\quad [|SharedIF|] \\ \quad\quad\quad (B[\![exec.e.IFb.f \leftarrow exec.e.IFa.f \\ \quad\quad\quad\quad\quad \mid (IFa, IFb, Fns) \in Connections, \\ \quad\quad\quad\quad\quad\quad e \in Exec, f \in Fns]\!])) \end{array}$$

where *Connections* is a set of 3-tuples specifying the interfaces to be connected. The *wiring* process renames *exec* events for *IFb.f* to make them appear as the corresponding *IFa.f* events, and synchronizes the components *A* and *B* on the shared *IFa.f* events.

Multiple-wiring of interfaces results in fan-in of calls to the component that is multiply-connected, and fan-out of calls from that component. Fan-in is well-modeled by using a multiple-renaming scheme. Modeling fan-out is slightly more complex, since it is necessary to enforce serial execution of the fan-out functions. Our approach to modeling fan-out is to insert an intermediate process between the caller and the functions called in the fan-out. This approach resembles the nesC compiler's use of intermediate functions to implement fan-out [9]. Space constraints preclude presenting the fan-out intermediate process.

C. Easing the Translation

The mapping described above provides a way to model nesC programs in CSP, but could be easier to use. To clarify the relationship between nesC programs and our CSP model, we define several auxiliary processes that overlay the basic process model with a "syntactic sugar" that more closely matches nesC syntax. For synchronous functions and calls, we define the processes

$$\begin{array}{l} command(f, Body) = FnDef(f, Body) \\ event(f, Body) = FnDef(f, Body) \\ call(f) = FnExec(f, sync) \\ signal(f) = FnExec(f, sync) \end{array}$$

which allow us to translate the *SenseC* event

```
event void MilliTimer.fired()
{ call Read.read(); }
```

into the process

$$\begin{array}{l} event(SenseC\_MilliTimer.Timer\_fired, \\ \quad\quad call(SenseC.Read\_read)) \end{array}$$

Similarly, for asynchronous functions, and calls from asynchronous contexts, we define the processes

$$\begin{array}{l} async\_command(f, Body) = AsyncFnDef(f, Body) \\ async\_event(f, Body) = AsyncFnDef(f, Body) \\ async\_call(f) = FnExec(f, async) \\ async\_signal(f) = FnExec(f, async) \\ hw\_async\_signal(f) = FnExec(f, async) \end{array}$$

D. Scheduling and Preemption Models

TinyOS applications are nesC programs that run in the context of TinyOS. Given a model of a nesC program *AppComponentGraph*, we model the corresponding TinyOS application as the parallel composition

$$\begin{array}{l} Application(AppComponentGraph) = \\ \quad ((Configuration\_MainC \\ \quad [|\{|task\_post, task\_exec, \\ \quad\quad\quad exec.begin.MainC, exec.complete.MainC|\}|] \\ \quad AppComponentGraph) \\ \quad [|\{|start\_atomic, end\_atomic, task\_exec, exec|\}|] \\ \quad AsyncPreemption) \end{array}$$

in which *Configuration_MainC* and *AppComponentGraph* synchronize on all events prefixed by *exec.begin.MainC* or *exec.complete.MainC*, and all events associated with the *task_post* and *task_exec* channels, and both processes synchronize with *AsyncPreemption* on all events associated with the channels *start_atomic*, *end_atomic*, *task_exec*, and *exec*. The process *Configuration_MainC* is an abstract model of the TinyOS `MainC` component. As described below, it encapsulates the boot process, and the task scheduler. The process *AsyncPreemption*, also described below, models the relationship between synchronous and asynchronous execution in TinyOS. We developed these models based on existing informal descriptions of TinyOS execution semantics [8], and examination of the TinyOS 2.1 source code.

1) MainC and the Scheduler: The TinyOS `MainC` component provides platform and software initialization, and is also the component in which the TinyOS scheduler resides [8]. The `MainC` component has two interfaces, modeled in CSP as

$$\begin{array}{l} \text{datatype } BootIF = Boot\_booted \\ \text{datatype } InitIF = Init\_init \end{array}$$

$$\text{datatype } MainC\_IF = MainC.BootIF \\ \qquad\qquad\qquad | \; MainC\_SoftwareInit.InitIF$$

Within `MainC`, system startup activities are handled by `RealMainP`. Following the standard `RealMainP` module [11], we define *BootProcess* as

$$\begin{aligned}
&BootProcess = \\
&\quad atomic(call(MainC\_SoftwareInit.Init\_init)\ ; \\
&\qquad\qquad run\_sched \rightarrow SKIP)\ ; \\
&\quad signal(MainC.Boot\_booted)
\end{aligned}$$

Note that *BootProcess* does not model platform initialization, but does include software initialization. The meaning of the *atomic*() wrapper around software initialization will be clarified in the discussion of *AsyncPreemption*.

The *MainC* process model combines the boot process model with a model of the scheduler:

$$\begin{aligned}
&Configuration\_MainC = \\
&\ \ BootProcess \\
&\ \ [|\{exec.begin.MainC.Boot\_booted.sync, run\_sched\}|] \\
&\ \ Scheduler
\end{aligned}$$

The interface between *BootProcess* and *Scheduler* is used to prevent tasks from being executed during software initialization, and to prevent the boot process from signaling *MainC.Boot_booted* until the scheduler has completed executing any tasks posted during initialization.

The *Scheduler* process models the standard TinyOS FIFO scheduler. In TinyOS 2.x, posting a task to the scheduler when it is already enqueued has no effect [8]. Our model reflects this behavior.

Internally, the *Scheduler* model is split into several processes, each representing a different scheduler state:

$$\begin{aligned}
&Scheduler = \\
&\quad \text{let} \\
&\qquad SchInit(Q, Postable) = \cdots \\
&\qquad SchNext(\langle\rangle, \_) = \cdots \\
&\qquad SchNext(\langle t\rangle \frown Q, Postable) = \ldots \\
&\qquad SchExec(Q, Postable, t) = \cdots \\
&\quad \text{within} \\
&\qquad SchInit(\langle\rangle, Task)
\end{aligned}$$

In the initial state, tasks may be posted to the scheduler, but are never executed. Tasks are only added to the task queue if they are in the set of *Postable* tasks that have not yet been enqueued. The scheduler transitions to a state in which tasks can be executed when the *run_sched* event occurs.

$$\begin{aligned}
&SchInit(Q, Postable) = \\
&\quad (run\_sched \rightarrow SchNext(Q, Postable)) \\
&\quad \Box\ (task\_post?t : Postable \rightarrow \\
&\qquad\quad SchInit(Q \frown \langle t\rangle, Postable \setminus \{t\})) \\
&\quad \Box\ (task\_post?t' : (Task \setminus Postable) \rightarrow \\
&\qquad\quad SchNext(Q, Postable))
\end{aligned}$$

If the task queue is empty, then any task can be posted. In addition, the empty-queue state is the only one in which the scheduler permits the *MainC.Boot_booted* event to proceed, thus ensuring that tasks posted during software initialization are cleared before the boot signal is sent.

$$\begin{aligned}
&SchNext(\langle\rangle, \_) = \\
&\quad (task\_post?t \rightarrow SchNext(\langle t\rangle, Task \setminus \{t\})) \\
&\quad \Box\ (exec.begin.MainC.Boot\_booted.sync \rightarrow \\
&\qquad\quad SchNext(\langle\rangle, Task))
\end{aligned}$$

If the task queue is not empty, the task at the head of the queue is executed. New tasks may also be posted. Note that a task becomes postable as soon as it has started executing. This allows tasks to post themselves.

$$\begin{aligned}
&SchNext(\langle t\rangle \frown Q, Postable) = \\
&\quad (task\_exec.begin!t \rightarrow \\
&\qquad SchExec(Q, Postable \cup \{t\}, t) \\
&\quad \Box\ (task\_post?t' : Postable \rightarrow \\
&\qquad\quad SchNext(\langle t\rangle \frown Q \frown \langle t'\rangle, Postable \setminus \{t'\})) \\
&\quad \Box\ (task\_post?t'' : (Task \setminus Postable) \rightarrow \\
&\qquad\quad SchNext(\langle t\rangle \frown Q, Postable))
\end{aligned}$$

Finally, when the scheduler is executing a task, it waits for the task to complete before returning to a state in which it is ready to execute another task.

$$\begin{aligned}
&SchExec(Q, Postable, t) = \\
&\quad (task\_exec.complete.t \rightarrow SchNext(Q, Postable)) \\
&\quad \Box\ (task\_post?t' : Postable \rightarrow \\
&\qquad\quad SchExec(Q \frown \langle t'\rangle, Postable \setminus \{t'\}, t)) \\
&\quad \Box\ (task\_post?t'' : (Task \setminus Postable) \rightarrow \\
&\qquad\quad SchExec(Q, Postable, t))
\end{aligned}$$

2) Preemption: The *AsyncPreemption* process models preemption of synchronous execution. It does this by blocking the events associated with synchronous functions whenever an interrupt event occurs. We split *AsyncPreemption* into two execution states, which represent synchronous and asynchronous execution:

$$\begin{aligned}
&AsyncPreemption = \\
&\quad \text{let} \\
&\qquad SyncExec(inAtomic) = \cdots \\
&\qquad AsyncExec(\{\}, inAtomic) = \cdots \\
&\qquad AsyncExec(Active, inAtomic) = \cdots \\
&\quad \text{within} \\
&\qquad SyncExec(false)
\end{aligned}$$

Each execution state places constraints on the functions that the application can execute. To apply these constraints, *AsyncPreemption* process synchronizes with the scheduler and application models on all events associated with the *task_exec* and *exec* channels, i.e., $AllActions = \{|task\_exec, exec|\}$. Because tasks can be posted from an asynchronous context *AsyncPreemption* does not place any constraints on *task_post* events. We divide the events in *AllActions* into three categories: *hw_async* events that represent interrupts, function calls made from asynchronous contexts, and events that are permissible in a synchronous context and don't indicate an interrupt:

$$\begin{aligned} HwActions = \{&exec.e.f.hw\_async \\ &\mid e \in Exec, f \in NesC\_Function\} \\ AsyncActions = \{&exec.e.f.async \\ &\mid e \in Exec, f \in NesC\_Function\} \\ SyncActions = \ &AllActions \setminus HwActions \end{aligned}$$

Within both execution states, the application can prevent preemption from occurring by declaring an *atomic* block. The transition into and out of an atomic block is communicated to the *AsyncPreemption* model via the events *start_atomic* and *end_atomic*. These events are used in the application model to define an atomic block via the process

$$\begin{aligned} atomic(Block) = \ &start\_atomic \rightarrow Block \ ; \\ &end\_atomic \rightarrow SKIP \end{aligned}$$

The synchronous execution state permits both task execution and execution of any function. If the application is not in an atomic block then interrupt events may occur, in which case the appropriate interrupt handler is invoked, and execution transitions to the asynchronous state.

$$\begin{aligned} &SyncExec(inAtomic) = \\ &\quad (\square\, s : SyncActions \bullet s \rightarrow SyncExec(inAtomic)) \\ &\quad \square\ (\neg\, inAtomic \ \&\ exec.begin?i : IntHandlers \rightarrow \\ &\qquad\qquad AsyncExec(\{i\}, false)) \\ &\quad \square\ (start\_atomic \rightarrow SyncExec(true)) \\ &\quad \square\ (end\_atomic \rightarrow SyncExec(false)) \end{aligned}$$

The set $IntHandlers \subseteq HwActions$ is the set of all interrupt-handling functions included in the application.

In the asynchronous execution state, *AsyncPreemption* is only prepared to proceed with those events that are permitted in an asynchronous context. Again, if the application is not in an atomic block then interrupt events may occur, in which case the new interrupt is added to the set of active interrupts. When all active interrupt handlers have completed, execution returns to the synchronous state.

$$\begin{aligned} &AsyncExec(\{\}, inAtomic) = SyncExec(inAtomic) \\ &AsyncExec(Active, inAtomic) = \\ &\quad (\square\, a : AsyncActions \bullet a \rightarrow \\ &\qquad AsyncExec(Active, inAtomic)) \\ &\quad \square\ (exec.complete?i : Active \rightarrow \\ &\qquad AsyncExec(Active \setminus \{i\}, inAtomic)) \\ &\quad \square\ ((\neg\, inAtomic \wedge \#Active < MAX)\ \& \\ &\qquad exec.begin?i : (IntHandlers \setminus Active) \rightarrow \\ &\qquad\quad AsyncExec(Active \cup \{i\}, inAtomic)) \\ &\quad \square\ (start\_atomic \rightarrow AsyncExec(Active, true)) \\ &\quad \square\ (end\_atomic \rightarrow AsyncExec(Active, false)) \end{aligned}$$

Note that the model presented here does not support nested atomic blocks (these are typically optimized away by the nesC compiler [8]), and allows multiple interrupt-handlers to interleave their actions.

IV. Example: RadioSense Application

Now that we have both a mapping from nesC programs to CSP processes, and a model of TinyOS scheduling and preemption, we're ready to look at an example CSP model of a TinyOS application, and how that model can be used to analyze the component interactions within the application. As our example application, we use `RadioSense`, a stripped-down version of the `RadioSenseToLeds` application distributed with TinyOS. The `RadioSense` application eliminates the packet reception and LED control functions found in `RadioSenseToLeds`, and performs only the sensing and radio transmission functions. Our analysis of the application focuses on checking that it performs these functions, and does so correctly.

A. Model

As with any modeling effort, we make some simplifying assumptions to ease model construction:

1) We assume that it is always possible to create a packet.
2) We use abstract models of the TinyOS components that are connected to the core application module. These models approximate the behavior of the actual TinyOS components, but avoid modeling complex internal implementation details or virtualization layers.

Even with these simplifying assumptions the complete model is too large to present here. We have made the full CSP model available at *http://coweb.elec.canterbury.ac.nz/cda/uploads/tos-app-exmpl.csp*, and present only the top-level application model in full detail here.

Figs. 2 and 3 show the top-level configuration of the `RadioSense` application, and an abbreviated version of the core module. The `RadioSenseC` module periodically reads a sensor and transmits the sensed data, using the state variable `locked` to track whether the radio is in use. The CSP process models corresponding to the nesC code appear in Figs. 4 and 5. Note that *MainC* is not included in the configuration model, because it is already part of our TinyOS model, and that *AMSenderC* is wired to *ActiveMessageC* in an approximation of the actual TinyOS implementation.

Since our analysis of the `RadioSenseC` application is focused on data sensing and transmission, it is useful to understand something about how data transmission is modeled. Transmission is initiated by calling the *send* command on the *AMSenderC* component, which relays the command down to *ActiveMessageC*. Beneath the *ActiveMessageC* component is a simple radio behavior model that abstracts from the details of medium-access and message-content, and uses *tx* events to represent the start and end of externally observable

```
configuration RadioSenseAppC {}
implementation {
  components MainC, RadioSenseC as App;
  components new DemoSensorC();
  components ActiveMessageC;
  components new AMSenderC(...);
  components new TimerMilliC();

  App.Boot -> MainC.Boot;
  App.AMSend -> AMSenderC;
  App.RadioControl -> ActiveMessageC;
  App.MilliTimer -> TimerMilliC;
  App.Packet -> AMSenderC;
  App.Read -> DemoSensorC;
}
```

Figure 2. RadioSenseAppC configuration

```
module RadioSenseC {
  uses {
    interface Boot;
    interface AMSend;
    interface Timer<TMilli> as MilliTimer;
    interface Packet;
    interface Read<uint16_t>;
    interface SplitControl as RadioControl;
  }
}
implementation {
  ...
  bool locked = FALSE;
  event void Boot.booted()
  { call RadioControl.start(); }
  event void RadioControl.startDone(...)
  { call MilliTimer.startPeriodic(250); }
  event void RadioControl.stopDone(...) {}
  event void MilliTimer.fired()
  { call Read.read(); }
  event void Read.readDone(...) {
    if (locked) {
      return;
    } else {
      ...
      call AMSend.send(...);
      locked = TRUE;
    }
  }
  event void AMSend.sendDone(...)
  { locked = FALSE; }
}
```

Figure 3. RadioSenseC module

$$
\begin{array}{l}
\mathit{Configuration\_RadioSenseAppC} = \\
\quad \text{let} \\
\qquad \mathit{Components} = \\
\qquad\quad (\mathit{Module\_TimerC} \mathbin{|||} \mathit{Module\_DemoSensorC}) \\
\qquad\quad \mathbin{|||} (\mathit{wiring}(\mathit{Module\_AMSenderC}, \\
\qquad\qquad\qquad \mathit{Configuration\_ActiveMessageC}, \\
\qquad\qquad\qquad \{(\mathit{AMSenderC\_AM}, \\
\qquad\qquad\qquad\quad \mathit{ActiveMessageC}, \\
\qquad\qquad\qquad\quad \mathit{AMSendIF})\})) \\
\quad \text{within} \\
\qquad \mathit{wiring}(\mathit{Module\_RadioSenseC}, \mathit{Components}, \\
\qquad\qquad \{(\mathit{RadioSenseC}, \mathit{AMSenderC}, \mathit{AMSendIF}), \\
\qquad\qquad\; (\mathit{RadioSenseC\_RadioControl}, \\
\qquad\qquad\quad \mathit{ActiveMessageC}, \mathit{SplitControlIF}), \\
\qquad\qquad\; (\mathit{RadioSenseC\_MilliTimer}, \\
\qquad\qquad\quad \mathit{TimerC}, \mathit{TimerIF}), \\
\qquad\qquad\; (\mathit{RadioSenseC}, \mathit{AMSenderC}, \mathit{PacketIF}), \\
\qquad\qquad\; (\mathit{RadioSenseC}, \mathit{DemoSensorC}, \mathit{ReadIF})\}) \\
\qquad [\![\mathit{exec.e.RadioSenseC.f.s} \leftarrow \mathit{exec.e.MainC.f.s} \\
\qquad\quad \mid e \in \mathit{Exec}, f \in \mathit{BootIF}, s \in \mathit{SyncType}]\!]
\end{array}
$$

Figure 4. RadioSenseAppC CSP model

radio transmissions. The radio has two states: idle and transmitting. If an attempt to send a new message is made while the radio is in the transmitting state, an error will occur. The radio behavior model is:

$$
\begin{array}{l}
\mathit{Module\_RadioHW} = \\
\quad \text{let} \\
\qquad \mathit{Idle} = \\
\qquad\quad \mathit{async\_command}(\mathit{RadioHW.Radio\_txStart}, \\
\qquad\qquad \mathit{tx.start} \rightarrow \mathit{SKIP});\ \mathit{Tx} \\
\qquad \mathit{Tx} = \\
\qquad\quad (\mathit{async\_command}(\mathit{RadioHW.Radio\_txStart}, \\
\qquad\qquad \mathit{tx.err} \rightarrow \mathit{SKIP});\ \mathit{STOP}) \\
\qquad\quad \Box\ (\mathit{tx.end} \rightarrow \mathit{hw\_async\_signal}(\\
\qquad\qquad\quad \mathit{RadioHW.Radio\_txDone});\ \mathit{Idle}) \\
\quad \text{within} \\
\qquad \mathit{Idle}
\end{array}
$$

Upon completion of a transmission, the *txDone* interrupt triggers a function in the *ActiveMessageP* module, which in turns posts a task to the scheduler.

$$
\begin{array}{l}
\mathit{TxDone} = \\
\quad \mathit{async\_event}(\mathit{ActiveMessageP.Radio\_txDone}, \\
\quad\;\; \mathit{post}(\mathit{ActiveMessageP\_task\_sendDone}));\ \mathit{TxDone}
\end{array}
$$

Execution of the task causes the *sendDone* event to be signaled to *AMSenderC*, and thence to *RadioSenseC*.

The sensor behavior model, like the radio model, is an abstract representation of sensor behavior. It is triggered by

$$
\begin{array}{l}
\mathit{Module\_RadioSenseC} = \\
\quad \mathsf{let} \\
\qquad \mathit{State}(\mathit{locked}) = \\
\qquad\quad (\mathit{event}(\mathit{RadioSenseC.Boot\_booted}, \mathit{call}(\mathit{RadioSenseC\_RadioControl.SplitControl\_start}));\ \mathit{State}(\mathit{locked})) \\
\qquad\quad \Box\ (\mathit{event}(\mathit{RadioSenseC\_RadioControl.SplitControl\_startDone}, \\
\qquad\qquad \mathit{call}(\mathit{RadioSenseC\_MilliTimer.Timer\_startPeriodic}));\ \mathit{State}(\mathit{locked})) \\
\qquad\quad \Box\ (\mathit{event}(\mathit{RadioSenseC\_RadioControl.SplitControl\_stopDone}, \mathit{SKIP});\ \mathit{State}(\mathit{locked})) \\
\qquad\quad \Box\ (\mathit{event}(\mathit{RadioSenseC\_MilliTimer.Timer\_fired}, \mathit{call}(\mathit{RadioSenseC.Read\_read}));\ \mathit{State}(\mathit{locked})) \\
\qquad\quad \Box\ (\mathit{event}(\mathit{RadioSenseC.Read\_readDone}, \\
\qquad\qquad \text{if } \neg\ \mathit{locked} \text{ then } (\mathit{call}(\mathit{RadioSenseC.AMSend\_send})) \text{ else } \mathit{SKIP});\ \mathit{State}(\mathit{true})) \\
\qquad\quad \Box\ (\mathit{event}(\mathit{RadioSenseC.AMSend\_sendDone}, \mathit{SKIP});\ \mathit{State}(\mathit{false})) \\
\quad \mathsf{within} \\
\qquad \mathit{State}(\mathit{false})
\end{array}
$$

Figure 5. RadioSenseC module CSP model

read commands from *RadioSenseC*, and posts a task which signals completion of the read operation:

$$
\begin{array}{l}
\mathit{task}(\mathit{DemoSensorC\_task\_senseResult}, \\
\quad \mathit{signal}(\mathit{DemoSensorC.Read\_readDone}))
\end{array}
$$

B. Analysis

We use FDR2 to check that the `RadioSense` application will sense and transmit data, and do so without generating errors. We express this check as an assertion that the application model refines an abstract process which is always ready to perform at least one of the events *tx.start*, *tx.end*, or *task_post.DemoSensorC_task_senseResult*, but will never perform *tx.err*:

$$
\begin{array}{l}
\mathit{DF}(A) = \sqcap\, a : A \bullet a \rightarrow \mathit{DF}(A) \\
E = \{| \mathit{tx}, \mathit{task\_post.DemoSensorC\_task\_senseResult} |\} \\
\mathsf{assert}\ \mathit{DF}(E \setminus \{\mathit{tx.err}\}) \sqsubseteq_{FD} \\
\quad \mathit{Application\_RadioSenseAppC} \setminus (\Sigma \setminus E)
\end{array}
$$

Checking the assertion with FDR2 confirms that the application model performs the sensing and transmission functions without error. Of course, a positive result is not all that interesting. As an example of error detection, suppose that a developer had decided to create a version of the `RadioSenseC` module that did not use the `locked` variable. Substituting a model of the alternative `RadioSenseC` in place of *Module_RadioSenseC*, and checking the assertion

$$
\begin{array}{l}
\mathsf{assert}\ \mathit{DF}(E \setminus \{\mathit{tx.err}\}) \sqsubseteq_{FD} \\
\quad \mathit{Application\_BadRadioSenseAppC} \setminus (\Sigma \setminus E)
\end{array}
$$

shows that the new module will cause an error to occur. Examination of the counterexample trace produced by FDR2 reveals the cause of the error: the new `RadioSenseC` module may try to transmit a second sensor reading before transmission of the first has completed. The analysis has thus detected a possible interleaving of data processing activities with radio transmission activities that produces undesirable application behavior, and has illuminated the rationale for including the `locked` variable in `RadioSenseC`.

V. Related Work

The work most closely related to ours is probably Rosa and Cunha's [12] effort to formalize nesC programs using LOTOS. Similar work by Völgyesi *et al.* [10] proposed the use of hierarchical interface automata to analyze nesC component interactions. Our CSP modeling approach draws inspiration from both efforts. It extends that earlier work by using a dual-event model of function-calls that gives a more accurate representation of nesC execution, and by adding models of TinyOS scheduling and preemption to facilitate analysis of their impact on application behavior.

Xie *et al.* [13] have proposed using the COSPAN model-checker for co-verification of TinyOS applications and platform hardware. Their approach to modeling TinyOS applications in the S/R language includes a model of TinyOS scheduling and preemption. However, their scheduler model differs from ours in that it directly controls the execution of each individual function-call, whereas our scheduler model more closely resembles the actual TinyOS task scheduler, which is responsible for coordinating task execution but does not directly control function-call sequencing.

Kothari *et al.* [14] recently developed a technique for extracting state-machines from TinyOS programs using symbolic execution. The resulting state-machines provide an abstracted view of the TinyOS program. In principle, symbolic execution could also be used to check the TinyOS program for desirable properties. However, doing so would require extending Kothari's method with some kind of specification language for defining the properties, and with a mechanism for checking those specifications during symbolic execution.

An alternative to static analysis of interaction models is runtime analysis of the component interactions themselves. To support such analysis, Archer *et al.* [15] recently proposed the addition of interface contract specifications to TinyOS applications. Contracts are checked at runtime, and can provide useful diagnostic information when an error is encountered. But, as with other testing techniques, interface contracts cannot provide any information about bugs that are not triggered by the testing regime.

VI. Conclusion

We have developed an approach for modeling TinyOS applications using the process algebra CSP. Our approach consists of a mapping from nesC to CSP, and a model of the TinyOS scheduling and preemption mechanisms.

Our approach to modeling and analyzing TinyOS applications is complementary to verification methods such as testing and runtime contract checks. Its benefits include:

- Analyses that can uncover unlikely, but undesirable, interleavings, allowing the discovery of low-probability bugs that might otherwise go undetected until the application has been widely deployed.
- Avoidance of the need to create explicit input scenarios, since model-checking involves exhaustive state-space exploration.
- Generation of counterexample traces when an error is detected, providing a clear indication of the sequence of commands and events that led to the error.
- The ability to evaluate new application designs, or proposed variations on existing designs, without going to the effort of creating a full implementation, and to carry out refinement-checking of application models against abstract models of network-level behavior.

Our modeling approach does have some limitations. It does not yet support representation of function arguments, or of parameterized interfaces. Both are a subject of future work. Nor does the model described herein support multiple simultaneous calls to the same function, although an extension that would do so appears to be relatively straightforward should it prove necessary to model that situation.

As with any application of model-checking, state-explosion is a concern. The refinement assertion in section IV-B takes $\sim$6 seconds to check on a 2.4 GHz laptop. Checking the same refinement assertion on a model that includes the packet reception and LED features of the full `RadioSenseToLeds` application takes $\sim$ 8 seconds, and requires the use of slightly more complicated scheduling and preemption models that provide identical behavior to those presented here, but are better tuned to the way FDR2 operates. The analysis of larger applications will inevitably require even more time, although predicting the exact amount of time is difficult since it depends on the structure of the application model. FDR2 does provide the ability to apply various state-space compression techniques [7], but we have not yet had the opportunity to evaluate their effectiveness for reducing the time required to analyze TinyOS application models. Ultimately, judicious use of abstraction is likely to be the key to successfully checking very large applications.

At present, our application models are constructed entirely by hand. We have begun work on a tool that will automate the generation of CSP models from nesC code. Our initial efforts are focused on translating just the nesC syntax, however it should also be possible to extract information from interface contracts to generate more constrained models that better reflect the intended use of components. Automated translation to CSP should make it easier to analyze existing applications for errors, which in turn would be helpful in identifying common errors that might be avoided through coding guidelines or additional compiler checks.

References

[1] P. Levis *et al.*, "TinyOS: An operating system for sensor networks," in *Ambient Intelligence*, W. Weber, J. Rabaey, and E. Aarts, Eds. Berlin: Springer, 2005, pp. 115–148.

[2] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 1–11, 2003.

[3] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire TinyOS applications," in *Proc. of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*. ACM, 2003, pp. 126–137.

[4] A. W. Roscoe, *The Theory and Practice of Concurrency*. Englewood Cliffs, NJ: Prentice Hall, 1998.

[5] M. Woehrle, C. Plessl, J. Beutel, and L. Thiele, "Increasing the reliability of wireless sensor networks with a distributed testing framework," in *Proc. of the 4th Workshop on Embedded Networked Sensors (EmNets '07)*. ACM, 2007, pp. 93–97.

[6] E. M. Clarke *et al.*, "Formal methods: state of the art and future directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, 1996.

[7] P. Gardiner *et al.*, *Failures-Divergences Refinement: FDR2 User Manual*, Formal Systems (Europe) Ltd, 2005.

[8] P. Levis, "TinyOS programming," June 2006, http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf.

[9] D. Gay, P. Levis, D. Culler, and E. Brewer, "nesC 1.2 language reference manual," August 2005, http://nescc.cvs.sourceforge.net/viewvc/*checkout*/nescc/nesc/doc/ref.pdf?revision=1.18.

[10] P. Völgyesi, M. Maróti, S. Dóra, E. Osses, and Á. Lédeczi, "Software composition and verification for sensor networks," *Science of Computer Programming*, vol. 56, no. 1-2, pp. 191–210, 2005.

[11] P. Levis, "TinyOS 2.x boot sequence," TinyOS Core Working Group, TinyOS Extension Proposal TEP-107, September 2007.

[12] N. S. Rosa and P. R. F. Cunha, "Using LOTOS for formalising wireless sensor network applications," *Sensors*, vol. 7, no. 8, pp. 1447–1461, 2007.

[13] F. Xie, G. Yang, and X. Song, "Compositional reasoning for hardware/software co-verification," in *Proc. of the 4th International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer, 2006, pp. 154–169.

[14] N. Kothari, T. Millstein, and R. Govindan, "Deriving state machines from TinyOS programs using symbolic execution," in *Proc. of the 7th International Conference on Information Processing in Sensor Networks (IPSN '08)*. IEEE Computer Society, 2008, pp. 271–282.

[15] W. Archer, P. Levis, and J. Regehr, "Interface contracts for TinyOS," in *Proc. of the 6th International Conference on Information Processing in Sensor Networks (IPSN '07)*. ACM, 2007, pp. 158–165.